CS162 - Artificial Intelligence
A* Search

Anna Pauxberger
28 February 2019

Find code and comments here:
https://gist.github.com/annapaux/8685e418a2088c7acc621cdf3a7046b1

Performance on Test Lists

Manhattan 2 extends Manhattan Distance by extending heuristic search to its children up until a specified depth. Is admissible since it uses Manhattan Distance as a base heuristic to calculate cost for the youngest children. It is always equal or greater than Manhattan distance.

	steps	frontier size	error code
Test List 1			

Hamming	29	20782	0
Manhattan	29	887	0
Manhattan 2	29	486	0
Test List 2			

Hamming	26	12330	0
Manhattan	26	957	0
Manhattan 2	26	682	0
Test List 3			

Hamming	18	420	0
Manhattan	18	51	0
Manhattan 2	18	46	0

annapaux / **eightpuzzle.py** Secret

Last active just now

eightpuzzle.py

```
"""
CS162 - Artificial Intelligence
A* Search
Anna Pauxberger
28 February 2019
"""


import numpy as np
import copy
from queue import PriorityQueue


class PuzzleNode:
    """A PuzzleNode represents the path to a current state, as well as the paths cost."""

    def __init__(self, state, fval, gval, parent=None):
        """Initializes a puzzle node. Takes as input the current state,  f-value, the g-value,
        and an optional parent node."""

        self.state = state                      # list of lists representing a n*n grid
        self.fval = fval                        # gval + hval (estimated heuristic value until the goal)
        assert isinstance(fval, int), fval      # ensure fval is an integer
        self.gval = gval                        # path cost until current state
        assert isinstance(gval, int), gval      # ensure gval is an integer
        self.parent = parent                    # any node except for initial state has another node as its parent
        self.pruned = False                     # if the node is pruned, there exists another node of the same state
                                                # with a lower gval

    def __lt__(self, other):
        """When two nodes are compared using 'less than', their f-val is compared. Used in PriorityQueue."""

        return self.fval < other.fval

    def __str__(self):
        """Prints the state of a node as a n*n grid."""

        grid_str = '\n'.join([' '.join([str(x) for x in line]) for line in self.state])
        return '-----\n' + grid_str + '\n-----'


def solvePuzzle(n, state, heuristic, prnt):
    """Solves an n*n puzzle. Verifies valid input and solvability. Returns number of steps, maximum frontier size
    of the priority queue and error rates (0: no error, -1: false input, -2: unsolvable).
    Input parameters: n is the size of the grid, state is a list of lists, heuristic is a heuristic function,
    and prnt is a boolean value that prints the optimal path solution, steps and frontier size."""

    if verify_state(n, state) is False:
        return 0, 0, -1

    if verify_solvability(n, state) is False:
        return 0, 0, -2

    # use numpy arrays for simple manipulation (e.g. reshape)
    goal = np.arange(n * n).reshape((n, n))
    initial = np.array(state)
    start_node = PuzzleNode(state=initial, fval=heuristic(initial), gval=0)

    # use a dictionary with a string of state as key and the unpruned node as value (gets overwritten when pruned)
```

```
    # string of initial state, since lists cannot be stored as dictionary keys
    state_to_unpruned_node = {str(initial): start_node}

    # frontier of nodes to expand next stored as priority queue where the lowest total cost (fval) has priority
    frontier = PriorityQueue()
    frontier.put(start_node)

    # number of total nodes expanded
    expansion_counter = 0

    # maximum frontier size of priority queue
    frontierSize = 0

    while not frontier.empty():

        # get node with highest priority (lowest fval) from the priority queue
        cur_node = frontier.get()

        # if the node is pruned, another more optimal node (lower gval) exists, therefore this node is neglected
        if cur_node.pruned:
            continue

        # if the state of the current node equals the goal state, break the while loop and print results
        if (cur_node.state == goal).all():
            break

        # define next states (see helper function)
        next_states = find_next_states(cur_node.state)
        for next_state in next_states:

            # add + 1 to gval to account for the additional step of moving from parent to child
            next_state_gval = cur_node.gval + 1

            # if the state of the child already exists in the dictionary: check for gval, else: continue to add it
            if str(next_state) in state_to_unpruned_node:

                # if the gval is smaller than the previously evaluated one, set 'pruned' of the node to True
                # since now a shorter path to this state has been identified, the previous path will no longer be
                # considered
                if next_state_gval < state_to_unpruned_node[str(next_state)].gval:
                    state_to_unpruned_node[str(next_state)].pruned = True
                else:
                    continue

            # add a 'next_node' for the 'next_state' and its path, and add it to the frontier priority queue
            hval = heuristic(next_state)
            next_node = PuzzleNode(state=next_state, fval=next_state_gval + hval, gval=next_state_gval,
                                   parent=cur_node)
            frontier.put(next_node)
            state_to_unpruned_node[str(next_state)] = next_node

        frontierSize = max(frontierSize, frontier.qsize())
        expansion_counter += 1

    # reconstruct the optimal path by referring to the parent of each node,
    # starting with the node that lead to the goal (saved as lists for testing script)
    optimal_path = [cur_node.state.tolist()]
    while cur_node.parent:
        optimal_path.append(cur_node.parent.state.tolist())
        cur_node = cur_node.parent

    # switch order of optimal path to represent initial state first and goal state last
    optimal_path = optimal_path[::-1]
    steps = len(optimal_path)
    err = 0

    if prnt:
        print(optimal_path)
```

```python
        print(steps)
        print(frontierSize)
        # print(expansion_counter)  # uncomment if run outside of testing script

    return steps, frontierSize, err


def find_next_states(current_state):
    """Finds next possible moves and evaluates which are valid (e.g. don't go beyond the grid). Returns list of
    possible valid next states."""

    zero_index = np.where(current_state == 0)
    row, column = (int(i) for i in zero_index)
    n = len(current_state)

    # possible moves are switches that are horizontally or vertically adjacent to the current empty zero-cell
    possible_moves = [(row - 1, column), (row + 1, column), (row, column - 1), (row, column + 1)]
    valid_moves = []
    for row, column in possible_moves:
        if 0 <= row < n and 0 <= column < n:
            valid_moves.append((row, column))

    next_states = []

    for move in valid_moves:
        new_state = copy.deepcopy(current_state)

        # switch the empty zero-cell with the value of an adjacent cell
        new_state[zero_index], new_state[move] = new_state[move], new_state[zero_index]
        next_states.append(new_state)

    return next_states


def memoize(h):
    """Memoization to remember the heuristic value determined by heuristic functions for given states.
     Used as decorator for heuristic functions to increase speed."""

    memo = {}

    def helper(state, *args):
        key = (str(state), tuple(args))
        if key not in memo:
            # *args allows for heuristic functions to have a different amount of variables
            memo[key] = h(state, *args)
        return memo[key]

    return helper


@memoize
def hamming_distance(state):
    """Heuristic function that counts the number of tiles that are in incorrect positions.
    Is admissible since a tile that is misplaced has to be moved at least once to reach the goal position."""

    n = len(state)
    state = np.array(state).reshape(-1)
    goal = np.arange(n * n)
    counter = 0

    for i in range(n * n):
        if state[i] != goal[i]:
            counter += 1

    return counter


@memoize
```

```python
def manhattan_distance(state):
    """Heuristic function that counts the vertical plus horizontal distance between a tile and its goal position.
    Is admissible since a tile, if no obstacle is in the way, has to move at least this path to reach the goal."""

    n = len(state)
    goal = np.arange(n * n).reshape((n, n))

    counter = 0

    # create a dictionary of where tiles are supposed to be
    goal_dict = {}
    for i in range(n):
        for j in range(n):
            goal_dict[goal[i][j]] = (i, j)

    for i in range(n):
        for j in range(n):
            if (state[i][j] != goal[i][j]) and (state[i][j] != 0):
                cur_val = state[i][j]
                goal_i, goal_j = goal_dict[cur_val]
                distance = abs(i - goal_i) + abs(j - goal_j)
                counter += distance

    return counter


@memoize
def manhattan_extended(state, depth=3):
    """Extends Manhattan Distance by extending heuristic search to its children up until a specified depth.
    Is admissible since it uses Manhattan Distance as a base heuristic to calculate cost for the youngest children. If
    depth was infinite, the heuristic would be perfect since it can find the optimal path to the goal.
    It is always equal or greater than Manhattan distance."""

    cur_hval = manhattan_distance(state)

    # when the goal is reached the heuristic value is 0, and we can return the value
    if cur_hval == 0:
        return 0

    # if depth is specified to be 0, the heuristic value is the manhattan distance
    if depth == 0:
        return cur_hval

    # recursively find the new_hval for the children of a node up until a specified depth
    # + 1 is added to account for the cost of adding a child to the previous path
    new_hval = min([manhattan_extended(child, depth - 1) for child in find_next_states(state)]) + 1

    return new_hval


def verify_state(n, state):
    """Verifies that the input state is a n*n matrix and that each number from 0 to n-1 is unique."""

    state = np.array(state)

    # verify shape (n,n)
    if state.shape != (n, n):
        return False

    # verify existence of all unique elements
    goal = [i for i in range(n * n)]
    original = sorted(state.reshape(-1))
    if original != goal:
        return False

    return True
```

```python
def verify_solvability(n, state):
    """If the Puzzle is solvable, the function returns True.
    See http://www.cs.princeton.edu/courses/archive/fall12/cos226/assignments/8puzzle.html"""

    board = [i for sublist in state for i in sublist]   # flatten the list
    board.remove(0)                                     # ignore blank zero state
    inversions = 0                                      # inversions are number of cases where list[i] < list[j]
                                                        # with i < j

    # count the number of inversions in the list
    for i in range(len(board)):
        for j in range(i + 1, len(board)):
            if board[j] < board[i]:
                inversions += 1

    # if board size is even: solvable if inversions + row number are uneven
    if n % 2 == 0:
        for i in range(n):
            for j in range(n):
                if state[i][j] == 0:
                    zero_row_index = i
        if (inversions + zero_row_index) % 2 != 0:
            return True

    # if board size is uneven: solvable if inversions are even
    else:
        if inversions % 2 == 0:
            return True

    return False


def main():
    n = 3
    test_lists = [[[5, 7, 6], [2, 4, 3], [8, 1, 0]],
                  [[7, 0, 8], [4, 6, 1], [5, 3, 2]],
                  [[2, 3, 7], [1, 8, 0], [6, 5, 4]]]

    heuristics = [hamming_distance, manhattan_distance, manhattan_extended]
    prnt = False

    for test_state in test_lists:
        for heuristic in heuristics:
            steps, frontierSize, err = solvePuzzle(n, test_state, heuristic, prnt)
            print(steps, frontierSize, err)
        print('-' * 15)


# define heuristics as a list for testing script
heuristics = [hamming_distance, manhattan_distance, manhattan_extended]


if __name__ == '__main__':
    main()
```